RESEARCH PROJECT REPORT

TURINGOL: A LANGUAGE FOR TURING MACHINES

by

G. F. Prentice

Computer Science Department

University of Canterbury

PREFACE:

This project was undertaken as an Honours 3 paper with the Department of Computer Science during the 1979 academic year. The knowledge, guidance and patience of Mr R. Harries, Project Supervisor, is gratefully acknowledged.

1.1 INTRODUCTION

Discussions of Turing computability often use a rather crude high level language to specify the actions of complex Turing Machines. This paper is a report on the design of a Turing Machine language and the implementation of a compiler for this language.

The classical Turing Machine is a device equipped with a two way infinite tape divided into squares. The machine can read or write characters from a finite alphabet on the tape in the square which is currently being scanned and it can move the scanning position to the left or right. The machine has a finite set of internal states. At any time, the action of the machine is determined by the character on the square under the read/write head and by the current state of the machine.

Such a machine is most often specified by a set of instructions of the form

"If in state Qi and scanning symbol Si then
Print symbol Sj
Move 1 square left (or right)
Transfer into state Qj "

which may be written as a quintuple (Qi Si Qj Sj D) where D, the direction is either left or right.

Automata (finite state machines, pushdown automata etc.) are often "programmed" using primitive instructions of a similar form to that described for Turing Machines. Church's thesis, [1], claims that any function for which there is an effective computational procedure may be computed by a Turing Machine but programs which do complicated things with only very primitive basic operations can involve a great amount of detail and complexity. These programs may be much more easily written as symbolic (or high level) programs [2]. Symbolic programs are easier to read and are less likely to contain careless errors.

Several attempts have been made to design programming languages for automata. Some of these are mentioned in the following section.

1.2 AUTOMATA PROGRAMMING LANGUAGES

A. [3] "A Turing Machine Simulator" describes a set of basic instructions for programming a Turing Machine Simulator. These are as follows:

RN: Shift the read-write head N squares to the right on the tape where N is an integer $0 \leq N \leq 100$

LN: Shift N squares to the left, $0 \leq N \leq 100$

W(X): Write X on the scanned square, where X is an element of the character set

T(α, X): Conditional transfer; if the scanned character is X, transfer control to the instruction whose symbolic address is α ; otherwise go to the next instruction in sequence.

T(α): Unconditional transfer to instruction at address α.

The language also includes the use of subroutines

A sample program is shown below. This machine prints a '1' on the first blank square at or to the right of the scanned square. The character 'B' represents a blank.

Symbolic address	Instruction
LOC1	T(LOC2, B)
	R1
	T(LOC1)
LOC2	W(1)
	END

The simulator makes two passes of the program (the first pass sets up symbol tables). Externally the simulator behaves like a Turing Machine, but its internal workings are quite different. For example, to move the read-write head 100 squares the simulator simply adds (or subtracts) 100 from a "location counter". A quintuple representation of the same action could require 100 different states.

B. [4] p. 137 describes a simple programming language for Turing Machines. An example program is shown here. This program prints a '1' on the first blank square at or to the right of the scanned square.

```
Tape alphabet is blank, one, .... ;
test:   if the tape symbol is "blank" then go to found;
        move right one square;
        go to test;
found:  print "one".
```

This language introduces a higher degree of symbolism and the program itself must be translated, possibly into quintuples or perhaps into something similar to the program in part A of this section (the Turing Machine simulator).

The production of code such as that understood by the Turing Machine simulator is an easier task than producing quintuples. For example, before a quintuple can be "written", three things must be known - the symbol to print, the direction to move in and the state to transfer into. The reading, printing, shifting and state transformation is performed in a single instruction. If code for the Turing Machine simulator is being produced then, for example, a GOTO statement simply translates into an unconditional transfer instruction T(∝). A 'print' statement translates into a W(X) instruction.

The high level language named Turingol (the name was borrowed from [5]) described in this report is compiled (translated) into a set of quintuples of the form { Qi Si Qj Sj D }. The reason for this was threefold.

A quintuple is the representation of an actual Turing Machine operation - it represents a single 'cycle' of the machine. Secondly, it is interesting to observe the number of states produced for a particular program. This can be taken as a measure of the complexity of the program. Thirdly, it presented an interesting challenge.

C. For the interested reader [1] page 90 describes a primitive language for Turing Machines.

2. THE TURINGOL LANGUAGE

2.1 INTRODUCTION

An example of a simple Turingol program (procedure) is given below. The procedure prints a '1' on the first non blank square or to the right of the scanned square.

```
PROCEDURE INTRO1;
TAPE SYMBOLS ARE '1', 'b', 'A';
START
WHILE NOT CURRENT = 'b' DO
        MOVE RIGHT 1
WEND
PRINT '1'
END
```

Another way of doing the same thing is

```
PROCEDURE INTRO2;
TAPE SYMBOLS ARE '1', 'b', 'A';
LABEL 10, 20;
START
10: IF CURRENT = 'b' THEN GOTO 20
                     ELSE MOVE RIGHT 1;
                          GOTO 10
                     FIN;
20: PRINT '1'
END .
```

The reserved word CURRENT is used to refer to the symbol currently under the read/write head.

The compiler produces exactly the same quintuples for each of these programs - these quintuples are presented here to help establish the relationship between a Turingol program and the machine code (namely quintuples) produced by the compiler.

Qi	Si	Qj	Sj	D
1	'1'	2	'1'	1
1	'A'	2	'A'	1
1	'b'	0	'1'	-
2	'1'	2	'1'	1
2	'A'	2	'A'	1
2	'b'	0	'1'	-

D = 1 is right
D = 0 is left
A final state of 0
halts the machine.

In fact only a single state other than 0 is required for this program - state '1' is produced by the compiler for every program and it is not always redundant. Every program begins executing in state 1.

The complete syntax of the language is given in Appendix A. The language may be divided into two parts: declarations and statements.

2.2 TURINGOL DECLARATIONS

Every program must begin with the line

PROCEDURE name;

This name may be used in a later program as the name of an external procedure. The declarations which follow must be in strict order.

TAPE SYMBOLS ARE '∝', 'β' ... ;

The machine will not recognize any other tape symbols.

Two types of variables may be declared and used in a program. The first type is BOOLEAN. A variable of this type takes the values true and false.

BOOLEAN FLAG: TRUE, STOP : FALSE;

Every declared BOOLEAN variable must be initialised to either true or false. The second type is CHAR. A variable of this type takes as its value any tape symbol.

CHAR ALPHA : 'A', CTROL : '$'

Every declared CHAR variable must be initialised with a particular tape symbol.

Any labels to be used in the program must be declared

```
LABEL 10, 20;
```

A label is an integer.

Finally, any external procedures the program wishes to use must be declared. An 'external procedure' is any previously compiled program that has been placed in the library (mentioned later).

```
PROCEDURE INTRO1 EXTERNAL;
PROCEDURE INTRO2 EXTERNAL;
```

Variable names and the program name may be of up to 9 alphanumeric characters.

The use of Boolean and Char variables is not essential for the description of any Turing Machine They are provided as a programmer convenience and should be used carefully. As will be seen, indiscriminate use of these variables could result in large numbers of states being produced.

2.3 TURINGOL STATEMENTS

Statements are of two types: those which describe an action to be performed by the Turing Machine and those which decide the flow of the program.

Statements of the first type are the MOVE, PRINT and HALT statements.

The halt statement is simply the word HALT. It will cause the Turing Machine to halt at the symbol currently being scanned. The print statement is the word PRINT followed by either a tape symbol or a variable of type CHAR.

```
PRINT '1';
```
- will print the symbol '1' on the square currently being scanned.

The MOVE statement has two forms

```
MOVE {LEFT } N;
     {RIGHT}
```
This moves the read/write head left or right N squares.

```
MOVE {LEFT } TO 'A', 'B';
     {RIGHT}
```
This moves the read/write head left or right until the symbol 'A' or the symbol 'B' is found.

There are some other features of this second form.

MOVE {LEFT / RIGHT} TO 'A'/'1', 'B'; The read/write head moves left or right until the symbol 'A' or the symbol 'B' is found. If the symbol 'A' is found, the symbol '1' is printed in its place.

Finally,

MOVE {LEFT / RIGHT} TO 'A'/'1', 'B'; 'C', 'D' ;

where 'A', 'B', 'C', 'D' and '1' are tape symbols. The symbols 'C' and 'D' after the colon are a direction to the compiler that the symbols 'C' and 'D' will not be found on the tape before either an 'A' or a 'B'. This will reduce the number of quintuples produced for this move statement by 2.

I. E. Assuming that the machine is in state 2 when it begins searching for an 'A' or 'B' to the right (what happens when machine reaches an 'A' or a 'B' is unknown as far as this MOVE statement is concerned). Then

Qi	Si	Qj	Sj	D
2	1	2	'1'	R
2	C	2	C	R
2	D	2	D	R
2	A	?	'1'	?
2	B	?	?	?

is reduced to

Qi	Si	Qj	Sj	D
2	'1'	2	'1'	R
2	A	?	'1'	?
2	B	?	?	?

A final word of explanation is needed for the MOVE statement such as

MOVE {LEFT / RIGHT} TO 'A', 'B'; The search for an 'A' or a 'B' begins at the square to the left or right of the square currently being scanned. This agrees with the intended spirit of the statement. Also, consider

MOVE RIGHT TO 'A', 'B';
MOVE LEFT TO 'B', 'C';
Assuming the tape symbols are 'A', 'B', 'C', 'D' and the machine is in state 2 then the quintuples produced are

Qi	Si	Qj	Sj	D	
2	'C'	2	'C'	R	Keep moving right until an 'A' or 'B'
2	'D'	2	'D'	R	is found.
2	'A'	3	'A'	L	The next state moves
2	'B'	3	'B'	L	left.
3	'A'	3	'A'	L	Keep moving left until a 'B' or 'C'
3	'D'	3	'D'	L	is found.
3	'B'	?	?	?	
3	'C'	?	?	?	

When state 2 finds an 'A' or a 'B' it transfers into state 3 but in the process the machine is required to move either left or right. Hence it arrives in state 3 looking at the square to the left of the 'A' or 'B' previously found.

2.4 There are six statement types that have no immediate relation to actual Turing Machine actions. These are the IF, WHILE, REPEAT, FOR, GOTO and assignment statements. The syntax of these statements is given in Appendix A. An IF statement is always terminated by the reserved word FIN. Similarly, WEND terminates a WHILE statement and FEND terminates a FOR statement. These statement terminators avoid the need for the BEGIN END compound statement. The body of a REPEAT statement is always executed at least once. The body of a WHILE statement may not be executed at all. The starting value of a FOR statement must always be 1. The priority of Boolean expression operators is

NOT 5
AND 4
OR 3

with parentheses having their usual meaning.

3 THE TURINGOL COMPILER

In this section the structures and functions of the actual compiler are presented. The compiler is written in Burroughs B6700 Pascal which was chosen in preference to ALGOL, firstly because it is less complicated and perhaps closer in spirit to the theme of the project, and secondly because the language Pascal appears to be coming into wide use. The Pascal typing facility has been used very little - variables and arrays are mostly of standard types.

3.1 The source program is translated into a sequence of tokens by the procedure GETOKEN which returns the next token each time it is called. A token consists of an integer code, which the compiler can recognize, and ten characters of semantic information which are either a string of alphanumeric characters (identifiers) or, a string of digits, or a character which is a tape symbol.

A Turingol program may be written in free format. A % sign will cause the remainder of the card to be ignored. Appendix B contains a table of tokens and their associated integer codes. There are 38 reserved words with codes 1 to 38. To determine whether a particular alphabetic string is a reserved word, the compiler checks only the reserved words which begin with the same letter as the string being checked. For this purpose a table of indexes (RSVINDX) into the reserved word array (RESERVED) is kept.

3.2 The task of transforming a Turingol program into a set of quintuples proceeds in three stages.

Stage 1 is a parse of the declarations of the Turingol program. This is performed by the procedure DECLARATIONS.

Stage 2 is a parse of the remainder of the Turingol program. This is performed by the procedures TRANSLATE and BVALUE. The procedure BVALUE specifically parses boolean expressions. The source program is translated and stored in an internal form.

Stage 3 produces quintuples using the information produced during stages 1 and 2. This stage is controlled by the procedure MAKEQUIN.

3.3 In order to explain the operation of the compiler it is necessary to explain what the compiler is required to do.

Consider the quintuples required to represent the following two statements, assuming the tape symbols are 'A', 'B', 'C'.

```
MOVE RIGHT TO 'A';
MOVE LEFT  TO 'B';
```

Qi	Si	Qj	Sj	D
2	B	2	B	R
2	C	2	C	R
2	A	3	A	L
3	A	3	A	L
3	C	3	C	L
3	B	-	-	-

Now suppose we have a boolean variable FLAG which may be either true or false and suppose that 10 is some location elsewhere in the program.

Then consider:

```
MOVE RIGHT TO 'A';
IF FLAG THEN GOTO 10 FIN;
MOVE LEFT  TO 'B'
```

With the previous quintuples, state 2 (the first move statement) always transferred into state 3 (the second move stmt). The requirement now is that the first move stmt only proceeds to the second move statement if the variable FLAG is false. The method of doing this is to duplicate the quintuples for the first move statement

Qi	Si	Qj	Sj	D						
2	B	2	B	R	4	B	4	B	R	
2	C	2	C	R	4	C	4	C	R	
2	A	3	A	L	4	A	-	-	-	branch to some state for location 10
3	A	3	A	L						
3	C	3	C	L						
3	B	-	-	-						

States 2 and 4 both represent the first MOVE stmt. State 2 'remembers' that the value of FLAG is false (and hence it transfers into state 3 which is the second move statement) while state 4 'remembers' that FLAG is true.

Suppose that we have two CHAR variables and 3 tape symbols, i.e. each variable could have any one of 3 values. There are nine possible combinations of the values of these variables (= 3 * * 2). The quintuples for any move stmt may need to be duplicated up to 9 times (i.e. 9 different

states), each state 'remembering' a particular combination of values for the variables. In general, if there are X char variables, Y tape symbols and Z boolean variables then the number of possible combinations is (Y * *X) * (2** Z). For example if there are 3 char variables, 2 boolean variables and 5 tape symbols, this value is 500!!

3.4 The move statement is the only statement which causes quintuples to be generated. There is a set of one or more states associated with each move statement. Before the quintuples of one state for a particular move statement can be completed the state to transfer into must be known (i.e, the 'next' move statement in the program). This 'next' move statement could be anywhere in the program and before it is 'arrived at' there may be some changes to the program variables (i.e. some assignment statements) so that the state produced at the next move statement 'remembers' a different combination of variable values to that 'remembered' by the previous state.

Consider a program with two boolean variables, FLAG1 and FLAG2 and tape symbols 'A', 'B', 'C'.

```
        FLAG1: = FALSE:
        FLAG2: = TRUE;
10:     MOVE RIGHT TO 'A';
        IF FLAG1 THEN MOVE LEFT TO 'B'
        ELSE
        IF FLAG2 THEN MOVE RIGHT TO 'C'
        ELSE MOVE LEFT TO 'A' FIN
    FIN;
*   FLAG1: = NOT FLAG1;
    IF FLAG1 THEN GOTO 10; FIN;
```

The first time the move statement at label 10 is 'arrived at', the values of FLAG1 and FLAG2 are false and true respectively so a state is produced which 'remembers' this combination of variable values. The value of FLAG1 is complemented at the statement labelled with * (so, first time through, it will change from false to true) and the final if statement will cause a branch back to label 10. Hence we need another state for the move statement at label 10 which remembers the values of FLAG1 and FLAG2 as both being true.

The state at label 10 for which FLAG1 is true must transfer into a state which represents the MOVE LEFT TO 'B' statement. The state

at label 10 for which FLAG1 is false and FLAG2 is true must transfer into a state which represents the MOVE RIGHT TO 'C' statement. Since FLAG2 is never false there is no state or quintuples produced for the MOVE LEFT TO 'A' statement, and thus the compiler is, to some extent, optimising. The quintuples produced would be

2	B	2	B	R	MOVE RIGHT	4	B	4	B	R	MOVE RIGHT
2	C	2	C	R	TO 'A';	4	C	4	C	R	TO 'A'
2	A	3	A	R	(FLAG1 = false)	4	A	5	A	R	(FLAG1 = true)
3	A	3	A	R	MOVE RIGHT	5	A	5	A	L	MOVE LEFT
3	B	3	B	R	TO 'C';	5	C	5	C	L	TO 'B'
3	C	4	C	R	(FLAG1 = false)	5	B	-	-	-	(FLAG1 = true)

States 2 and 4 are associated with the move statement at label 10.

It is convenient at this point to explain why integer variables are not included in the language. Suppose the language included integer variables. Then, if I is an integer variable, consider

```
    I: = 0;
10: MOVE RIGHT TO 'A', 'B';
    IF CURRENT = 'A' THEN
        I: = I + 1;
        GOTO 10 FIN;
```

The program would be expected to count the number of times the symbol 'A' occurs before the symbol 'B' occurs. This is a potentially infinite value. The program needs a new state at label 10 for each different value of the variable I (to 'remember' the value of I). The number of states required depends on the tape input and unless upper and lower bounds are placed on the allowable values of I, there is no static representation for the program.

Now consider the MOVE statement

```
MOVE RIGHT TO 'A', 'B';
```

with three tape symbols 'A', 'B', 'C' there would be three quintuples produced

```
2 C 2 C R
2 A - - - transfers to some new state  } possibly two
2 B - - -    "    "   "   "    "     } different states
```

The second of these quintuples is applicable if the symbol 'A' is under the read-head, the third if 'B' is under the read-head. The next move statement reached in the program is dependent on the symbol currently under the read-head, i.e. the value of CURRENT.

For example

(i) IF CURRENT= 'A' THEN GOTO 10
ELSE MOVE LEFT TO 'C' FIN;

or (ii) If ALPHA is a CHAR variable
ALPHA: = CURRENT

will assign a different value to ALPHA for the two cases. (I. E. either the symbol 'A' or the symbol 'B' - assuming no PRINT statement occurs before the assignment statement.)

The point of this discussion is to show that there may be several sets of quintuples produced for a particular MOVE statement. The quintuples within a set all have the same initial state, but different sets have different initial states. The quintuples within each of these sets may correspond to branches to different locations in the program - possibly even back to the same MOVE statement. The task of the compiler is to produce this network of quintuples.

Consider the following hypothetical program with tape symbols 'A', 'B', 'C'.

```
1:          ALPHA : = 'C' ;
2:     10:  MOVE RIGHT TO 'A', 'B';
3:          IF ALPHA = 'C' THEN ALPHA: = CURRENT FIN;
4:          IF ALPHA = 'B' THEN GOTO 10 FIN;
5:          MOVE RIGHT TO 'B', 'C' ;
6:          ALPHA : = CURRENT ;
7:          GOTO 10;
```

When the compiler first encounters the move statement in line 2 the value of ALPHA is 'C'. At this stage the compiler does not know whether any more states will be needed for this MOVE statement to remember the other possible values of ALPHA. The compiler finds out what states are needed for this MOVE statement by working through the program.

It begins by taking the first of the two cases for the MOVE statement in line 2 (i.e. the case when the symbol 'A' is found. It returns to

the second case later by making an entry in a stack.) State 2 will represent the MOVE statement at line 2 with the value of ALPHA being 'C'. At line 3 the value of ALPHA is 'C' so the assignment statement is applied - the value of CURRENT is 'A' so ALPHA becomes 'A'. Line 4 has no effect.

A new state is created for the MOVE statement at line 5 - state 3. A quintuple for the most recent MOVE statement is printed. I.E. {2 'A' 3 'A' R} . There are two cases to consider for the MOVE statement at line 5 so a stack entry is made and the first case (CURRENT = 'B') is expanded. The value of ALPHA becomes 'B' at line 6. The GOTO 10 at line 7 sends the compiler to the MOVE statement at line 2. A new state is created, state 4, which 'remembers' the value of ALPHA as 'B' for this MOVE statement (A quintuple for the most recent MOVE statement is created {3 B 4 B R} .

The expansion of state 4 involves two cases (symbols 'A' and 'B') so a stack entry is made and the first case (CURRENT = 'A') is expanded. Line 3 has no effect. At line 4 the value of ALPHA is 'B' so the GOTO 10 statement is applied - the compiler sees the MOVE statement at line 2 again. The compiler discovers that state 4 was produced for this MOVE statement to remember the value of ALPHA as 'B' so it does not produce a new state. The expansion halts at this point. A quintuple is produced - {4 A 4 A R}.

The most recent stack entry is now considered - this is the entry made when state 4 was created at line 2. The stack entry is such that it enables the values of the program variables to be restored (i.e. ALPHA = 'B'). The second of the two cases (CURRENT = 'B') is expanded and after lines 3, 4 and 2 the quintuple {4 B 4 B R} is printed.

The most recent stack entry is again considered - this is the entry made when state 3 was created at line 5. ALPHA is restored to 'A', CURRENT = 'C' - so at line 6 ALPHA will become 'C' and at line 2 the expansion will halt (because state 2 represents the value of ALPHA as 'C' for this MOVE statement) and the quintuple {3 C 2 C R} is printed.

This process continues until the stack is empty. The result is

state 2 ALPHA = 'C' (line 2)
3 ALPHA = 'A' (line 5)
4 ALPHA = 'B' (line 2)

and

```
2 C 2 C R    3 A 3 A R    4 C 4 C R
2 A 3 A R    3 B 4 B R    4 A 4 A R
2 B 4 B R    3 C 2 C R    4 B 4 B R
```

The expansion is initiated from state 1 - from state 1 the expansion for each declared tape symbol proceeds separately. For the example above

```
1 A 2 A R      so state 1 is redundant here.
1 B 2 B R
1 C 2 C R
```

It is important that the values of the program variables are always known. For this reason all program variables must be assigned initial values when they are declared.

The REPEAT and WHILE statements present no difficulties. At the end of a repeat statement (i.e. UNTIL bool expr) the effect is the same as if there was an IF NOT (bool expr) THEN GOTO - statement. Similarly for the WHILE statement.

The representation of FOR statements is as follows. A new state is created whenever there is a new value of the FOR loop counter to be 'remembered'. The expansion proceeds as before but halts only when a MOVE statement is reached that 'remembers' all the same program variable values as the current expansion and also the same value of the FOR loop counter. When continuing the expansion from the most recent stack entry, the value of the FOR loop counter is restored along with the program variables.

Nested FOR statements are not allowed because they require the checking and restoring of an arbitrary number of FOR loop counters - possible but not justified. The FOR statement is useful enough as it stands.

If the reader is concerned that the expansion process described above might continue ad infinitum be reassured that it doesn't. The number of states produced for a particular MOVE statement is never more than the number of different combinations of program variable (including FOR loop variable) values. This may be large but it is finite.

However, an infinite loop may easily be programmed into the Turing Machine. The Turing Machine for the hypothetical program above never halts and once in state 4 it will never leave.

An infinite loop such as

```
REPEAT
    FLAG: = FALSE
    UNTIL FLAG
```

is treated as an error by the compiler when it tries to 'execute' it in the process of getting to the 'next' MOVE statement.

4.1 Stage 1 of the compiler is a parse of the Turingol declarations, performed by the procedure DECLARATIONS.

A. Tape Symbols - each declared tape symbol occupies one element of the array SYMBOLS (array [1 ... 64] of CHAR). NUMSYM is set to the number of declared symbols.

B. Program identifiers - either boolean or char variables or external procedure names are written into the array IDENTIFIERS ([1 ... 25] of packed array [1 ... 10] of CHAR). Boolean variables are first - NUMBOOL is the number of boolean variables; char variables next - NUMCHAR is the number of char variables; external procedure identifiers are last - NUMPROC is the number of these.

At any time during the generation of quintuples, the values of the program variables are contained in the arrays BOOL ([1 ... 10] of Integer) and CHARS ([1 ... 10] of integer). If the associated boolean variable is true the value in BOOL is 1, otherwise 0. The value of a CHAR variable is an element of the SYMBOLS array and its index in this array is contained in the appropriate element of the CHARS array (e.g. if there are two CHAR variables, CHARS 1 and 2 are used). The arrays BOOL and CHARS are set up with the initial values of the program variables.

C. Labels - the integer value of each declared label is placed in the first element of a row of the array LABELS ([1 ... 10] of array [1 ... 2] of integer). The 'location' of the label is later placed in the second element when the location is known (during procedure TRANSLATE).

D. External procedures - the name of the external procedure is located in the library index which gives the location (index) in the library of the quintuples for the procedure. This index is placed in the first element of a row of the array EXTPROC ([1 ... 10] of array [1 ... 2] of integer). The second element is an index into the array QUINSYM (array [1 ... 300] of integer), which contains information involving tape symbols used when the procedure is bound in.

This procedure will also discover any syntax mistakes and set an error code (ECODE) appropriately.

4.2 To identify a particular combination of values of program variables easily, there is a linear mapping of these combinations onto the integers 0 to α where α is 1 less than the total number of combinations. For example, with two boolean variables F1 and F2, 1 char variable A1 and two tape symbols 'A', 'B' the mapping is

A1	F2	F1	Integer code
A	0	0	0
A	0	1	1
A	1	0	2
A	1	1	3
B	0	0	4
B	0	1	5
B	1	0	6
B	1	1	7

where 0 represents false
1 represents true

The procedure ENCODEST produces the integer code from the values of the program variables contained in the BOOL and CHARS arrays. The procedure DECODEST does the reverse of this. The integer code discussed here is referred to as the 'compiler state' when the compiler is producing quintuples because it represents the value of the program variables at any point in the expansion. When the compiler processes a MOVE statement it checks to see if there is already a TM (Turing Machine) state for the MOVE which represents the current value of the program variables. The list of states already produced for the MOVE forms a linked list in the array STATES (array [1 ... 1000] of array [1 ... 2] of integer). The index into this array is the TM state number. The first element in each row is the compiler state number and the second element points to the next node (TM state) in the list. The procedure STCHAIN checks this linked list, looking at the compiler state number in the first element of each node.

4.3 Stage 2 of the compiler parses the remainder of the program and translates it into an internal form which is used by the compiler at stage 3 to produce quintuples.

This internal form is contained in the integer array CTRLFILE (array [1 ... 150] of array [1 ... 4] of integer). The first element in each row is an integer number in the range 1 to 10 which is later used in the procedure MAKEQUIN as a case selector.

The procedure TRANSLATE produces the CTRLFILE array and checks the syntax at the same time. The syntax is checked using a 'recursive descent' technique. The recursion is handled by adding to a stack for each inward step of the recursion and chopping the stack for each outward step of the recursion. The stack involved is the array STAK ([1 ... 100] of integer). The procedure TRANSLATE is just a large case statement which is repeated until an error or the end of the program is found. The selector for the case statement is the top of the stack, STAK (pointed at by STKPTR). For example the stack index into the CTRLFILE of the first statement of a REPEAT is loaded onto the stack and after the matching UNTIL and boolean expression are found the location in the CTRLFILE to be branched to for the REPEAT is given by the value on the stack.

There is a rough correspondence between a statement in the source program and a row of the CTRLFILE. Each row of the CTRLFILE is loaded by the procedure LOADCTRL. The first element in a row of the CTRLFILE specifies what type of entry the row is. These are represented by the values 1 to 10.

1 Branch to some other location in the CTRLFILE

2 MOVE LEFT TO symbol

3 MOVE RIGHT TO symbol

4 MOVE {LEFT / RIGHT} N

5 Evaluate a boolean expression and branch to either the true or the false location

6 Assignment statement e.g. ALPHA: = 'A'

7 Print statement

8 Halt statement

9 For statement

10 Procedure call

These ten different entry types are sufficient to represent the source program in an internal form. IF, REPEAT, WHILE and GOTO statements are represented using entries of types 1 and 5.

As explained in section 3, the compiler has to work through the program expanding from MOVE statements; I. E. the compiler works through the CTRLFILE.

4. 4 The entries in the CTRLFILE form the basis of the compiler. These entries are explained in details in this section.

Type 1. A branch instruction. The second element in the row specifies a CTRLFILE index to be branched to. The third and fourth elements are used later to check for infinite loops.

Types 2 and 3:

$$\text{MOVE} \left\{ \begin{matrix} \text{LEFT} \\ \text{RIGHT} \end{matrix} \right\} \text{TO symbol/symbol, .. : symbol, symbol}$$

The second element in the row is an index into the array QUINSYM. The statement is allocated a 'block' in this array - the length of this block is equal to the number of tape symbols declared in the program. Suppose there are five tape symbols, 'A', 'B', 'C', 'D', 'E' which are at positions 1 to 5 respectively in the SYMBOLS array. Then for the statement

MOVE RIGHT TO 'A'/'D', 'B' : 'D'

the entries in QUINSYM are

A	B	C	D	E
4	2	0	-1	0

so that if the symbol 'A' is found the symbol 'D' (4) will be printed. This results in quintuples, say for states 8 and 9

8	C	8	C	R
8	R	8	E	R
8	A	9	D	-
8	B	9	B	-

A quintuple for the symbol 'D' is not generated (-1 in QUINSYM). Symbols 'C' and 'E' stay in the same state (0 in QUINSYM).

Type 4:

$$\text{MOVE} \begin{Bmatrix}\text{LEFT}\\ \text{RIGHT}\end{Bmatrix} \text{N} \quad \text{where N is an integer}$$

If N is 1, an entry of type 2 or 3 is made - suppose there are 4 tape symbols, 'A', 'B', 'C', 'D', then the entry represents

$$\text{MOVE} \begin{Bmatrix}\text{LEFT}\\ \text{RIGHT}\end{Bmatrix} \text{TO 'A', 'B', 'C', 'D'}$$ which causes the read/write head to move left or right one square.

If N is greater than 1, two entries are made. Firstly an entry of type 4 - the second element of this entry is 0 or 1 for the direction (left or right). The third element is used later as an index into the STATES array. The fourth element has the value N-1 which is the number of states produced to 'count' the number of squares moved. Each successive state moves the head one square left (or right). This entry is followed by an entry of type 2 or 3 which produces the Nth move of the head.

Type 5: Whenever a boolean expression is found it is converted to a string of integer codes in reverse polish form in the BOOLEXPR array ($[1 \ldots 400]$ of integer) by the procedure BVALUE. Later, during the production of quintuples, this integer string is evaluated (by procedure EVALBOOL) for the current values of the program variables and the result will be either true or false depending on the values of the program variables. The boolean expressions entered in the BOOLEXPR array are of variable length but there is no wasted space - a new expression begins immediately after the previous expression.

The second element in the row for the type 5 entry is an index into the BOOLEXPR array. The third element is the location to be branched to if the boolean expression is true (a CTRLFILE index) and the fourth element is the location for false.

For example

```
IF  bool expr.     THEN
    STMT 1;
    STMT2;
    STMT3     FIN;
STMT 4
```

The third element in the row is the CTRLFILE location of STMT1 and the fourth element in the row is the CTRLFILE location of STMT4.

Type 6: This is the entry made for an assignment statement. The second element is an integer code from 1 to 6 depending on the type of assignment. The procedure ASSIGN uses this code to actually do the assignment during stage 3 of the compiler. The third and fourth elements are either 'identifier numbers' (an index into the CHARS or BOOL arrays), a symbol number (an index into the SYMBOLS array), or a value of 0 or 1 representing true or false. The procedure ASSIGN should be looked at for the meaning of the six codes.

Type 7: This is the entry for a PRINT statement. The two cases are

(i) PRINT symbol - second element coded with 0
- third element is symbol number

(ii) PRINT identifier - second element coded with 1
- third element is identifier number

Type 8: This is the entry for a HALT statement. The entry contains no other information. When it is encountered during 'execution' of the CTRLFILE it causes the current expansion to halt. This is always the last entry in the CTRLFILE for every program.

Type 9: This is entered for a FOR statement. The second element in the entry specifies the number of times the FOR loop is to be executed. When the end of the FOR loop is reached, a type 1 entry is written into the CTRLFILE which causes a branch back to this type 9 entry. The third element in the entry specifies the CTRLFILE location to be branched to when the FOR loop has been completed.

The FOR statement also creates an entry in the array FORN (array [1 ... 10] of array [1 ... 4] of integer). The first and second elements in a row of this array are the CTRLFILE indexes of the start and end of the loop (i.e. the type 9 and type 1 entries). These two entries enable detection of a GOTO out of a FOR loop - which enables the special FOR loop counter variable (THISFOR) to be reset to zero. A GOTO into a FOR loop proceeds as if it were the zeroth time round the loop.

Type 10: This entry is made for a procedure call. The second element is an index into the array EXTPROC which contains the location of the procedure's quintuples in the library plus an index into the array QUINSYM.

5.1 Stage 3 of the compiler begins when the source program has been completely parsed and the CTRLFILE is complete. The procedure MAKEQUIN steps through the CTRLFILE, calling procedures MOVINSTR, EVALBOOL, ASSIGN and PROCEDURECALL when necessary. Section 3.5 explained the technique of producing states and quintuples from a Turingol program. Stage 3 of the compiler performs this expansion process. The stack involved is the array EXECSTAK ([1 ... 300] of array [1 ... 5] of integer). This stack is the basis of the expansion.

A move statement such as

MOVE RIGHT TO 'A', 'B';

with tape symbols 'A', 'B', 'C', 'D' could produce quintuples

2	C	2	C	R
2	D	2	D	R
2	A	3	A	R
2	B	4	B	R

The last two quintuples are expanded separately and will possibly transfer into different states. Because there are two separate expansions to be made, an entry is made in EXECSTAK. The second expansion is not begun until all paths leading from the first expansion have been covered.

Important variables for this expansion are

THISCURR	-	this is an index into the SYMBOLS array for the value of CURRENT
STATE	-	this is the number of the next Turing Machine state to be produced
ST	-	this is the 'compiler state number' or integer code representing the values of the program variables
KX	-	this is the index into the CTRLFILE
BS	-	this is the top of stack pointer for EXECSTAK
PRINT	-	this is an index into the SYMBOLS array for the symbol to be printed for the most recent quintuple being expanded
THISFOR	-	this is the value of the number of times round a FOR loop at the current point in the expansion.

As explained in Section 3.5, when a MOVE statement is encountered a list of states contained in the STATES array has to be checked to see if the

expansion should continue and create a new state. If the MOVE statement is outside a FOR loop, element 3 of the CTRLFILE entry for the MOVE statement is an index into the STATES array. I. E. a head of list pointer. If the MOVE statement is inside a FOR loop

```
E.G.  FOR I: = 1 TO 5 DO
            MOVE RIGHT TO 'A'
      FEND;
```

then there are five distinct linked lists associated with the MOVE statement. The head of list pointers for these are in the array STATHDR ([1 ... 400] of integer). Element 3 of the CTRLFILE entry for the move contains an index into the STATHDR array.

The same state checking and head of list pointer arrangements are used when a procedure call occurs in the CTRLFILE.

When quintuples are produced for a particular state, they are written into a random access file as contiguous records. Initially not all the quintuples for the state will be known, so a gap is left in the file and the records are filled in when the quintuples are known. When a MOVE statement is encountered in the CTRLFILE the most recent quintuple is written into the random access file by the procedure PRNTLAST.

Consider the MOVE statement

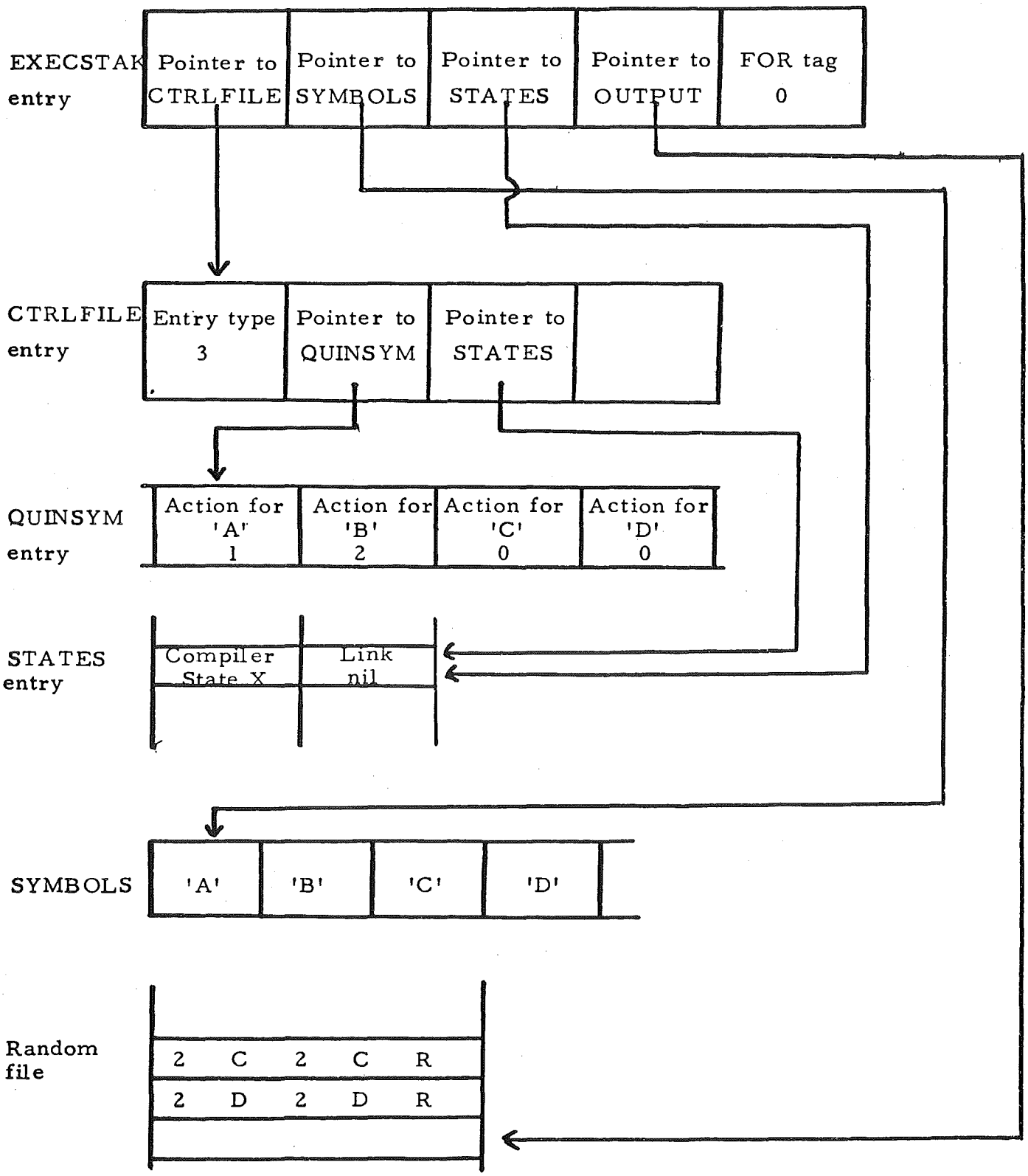
```
MOVE RIGHT TO 'A', 'B';
```

with tape symbols 'A', 'B', 'C', 'D' this could produce quintuples.

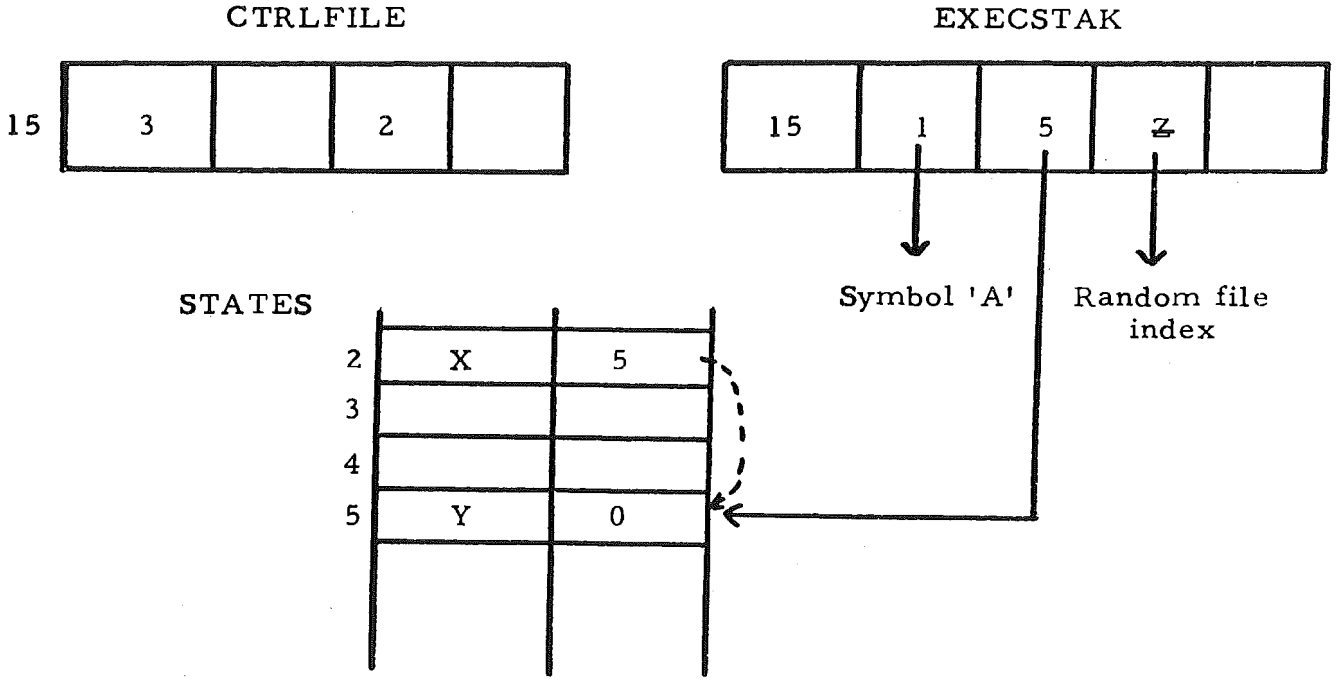
2	C	2	C	R
2	D	2	D	R
2	A	3	A	R
2	B	4	B	L

The first two quintuples are written to disk by the procedure PRNTTHIS when the MOVE statement is processed. The last two quintuples are not written into the random file immediately. The following page shows the state of the compiler when the third quintuple is being expanded. Referring to the next page, the value X in the STATES array is the 'compiler state number' associated with the MOVE statement - i. e. it specifies the values of the program variables. This number is used to restore the values of the program variables when backtracking occurs.

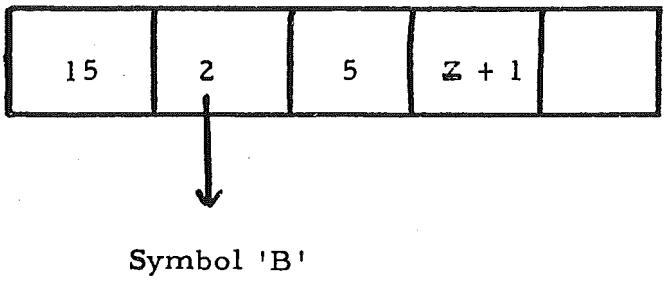
Control information associated with the 'A' part of MOVE RIGHT TO 'A', 'B'.



Suppose some time later we produce another state, say state 5, for this MOVE statement with internal state number Y, then the situation is



A new EXECSTAK entry has been made. When the quintuple for the symbol 'A' has been expanded the EXECSTAK entry is updated



5.2 THE LIBRARY IS A RANDOM ACCESS DISK FILE

Any Turingol program which compiles successfully has the quintuples written into this file. With each set of quintuples written into the file there is an associated list of state indexes. This list gives the random file index of each state in the program. For example, if there are five states there will be five records in this list, the first of which is the random file index (location) of state 1 etc. (This information is used by the simulator to locate the quintuples with a certain initial state).

The first record in the file specifies the number of programs (code files) contained in the library and also specifies the number of the next free record in the file. This is followed by 100 records which give the location of each code file in the library - one record for each code file. This code file index contains the program name followed by the random file index of the program's quintuples plus the random file index of the associated state indexes of the program.

5.3 THE BINDING IN OF PROCEDURES

If a Turingol program calls a procedure, this procedure (module) must be present in the library. Supposing the procedure has 3 states

1	A	2	A	R	
1	B	3	C	L	
1	C	"	C	R	
2	C	2	C	R	
2	A	0	A	-	halt
2	B	3	B	L	
3	B	3	B	L	
3	A	2	A	R	
3	C	0	C	-	halt

State 1 is never copied into the program but it specifies the state to be transferred into for each of the symbols 'A', 'B' and 'C'. E.G. the program is 'looking' at a 'B' when the procedure is called, the quintuple involved will transfer into state 3 or the procedure, print a 'C' and move left.

If the procedure halts in more than one quintuple, it will halt on a particular symbol and each of these halting quintuples is 'expanded' separately. (special EXECSTAK entry is made.) I.E. the exit from a procedure occurs at each halting quintuple.

Suppose the next state to be produced by the program was 5 when the procedure was called. Then states 2 and 3 of the procedure become states 5 and 6 of the program. The compiler copies all the quintuples for these two states into the new "codefile", changing the state numbers as it goes. The second state number of each halting quintuple is over-written with the random file index of the next 'halting quintuple' so that when the expansion of one 'halting quintuple' is complete the index of the next 'halting quintuple' to be expanded is available.

There must be some compatibility between the symbols of the main program and the symbols of the procedure. The requirement is that any symbol under the read/write head when the procedure call occurs must be recognized by the procedure (i.e. listed in state 1) and also that any symbol halted on by the procedure must be recognized by the main program. Hence the procedure may use symbols unknown in the main program and vice versa.

5.4 THE SIMULATOR

A Turing Machine simulator has been written (in Burroughs Pascal) which will execute the quintuples of a particular program. The tape input is supplied as a data deck of cards. The first card of these is not part of the tape. The first 9 characters on the first card are the name of the Turingol program to be run. The simulator will start the simulation scanning the first non blank character. If the tape is blank it will start scanning roughly in the middle of the tape.

The simulator writes out the tape input followed by the tape output once the Turing Machine halts.

6. CONCLUSIONS

There are many details that could not be included in this report. However, it is hoped sufficient detail has been included to provide a basic understanding of the compiler and what it does.

In retrospect there are changes and modifications which could be made to improve the Turingol language and also the compiler. For example, the compiler performs no particular optimisation other than that which is inherent in its design - further optimisation is certainly possible.

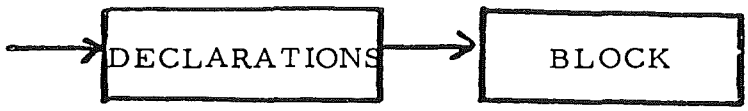
Hopefully the reader, if he has the chance, will feel motivated to write a Turingol program of his own.

APPENDIX A.

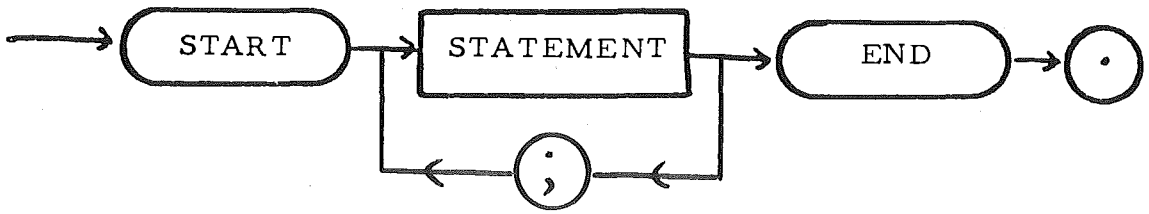
Turingol Syntax

The syntax of the Turingol language is specified by the following railroad diagrams. Any program which does not conform exactly to the given syntax will be rejected by the compiler. In such cases an error message will be produced giving some indication of the error.

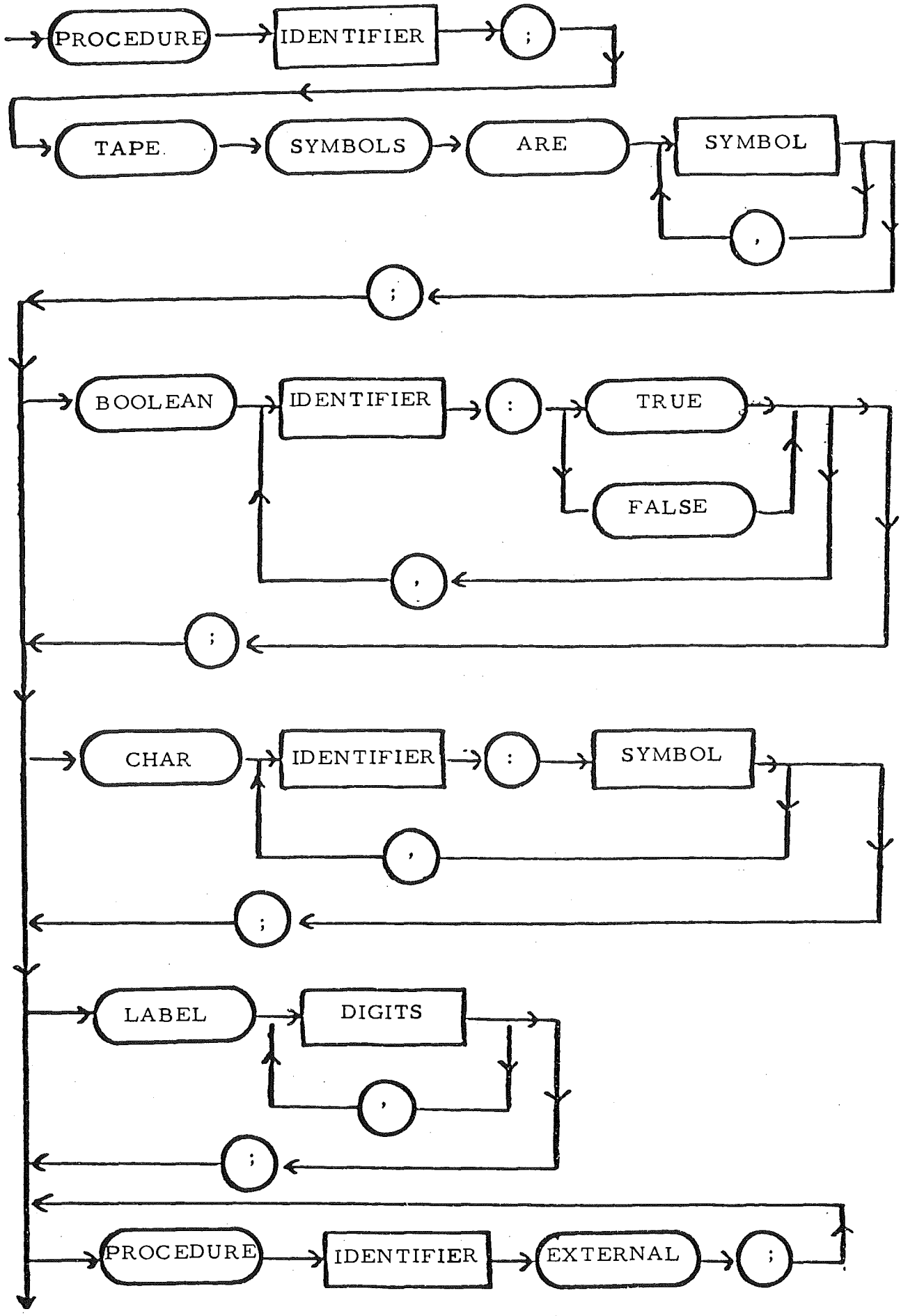
PROGRAM



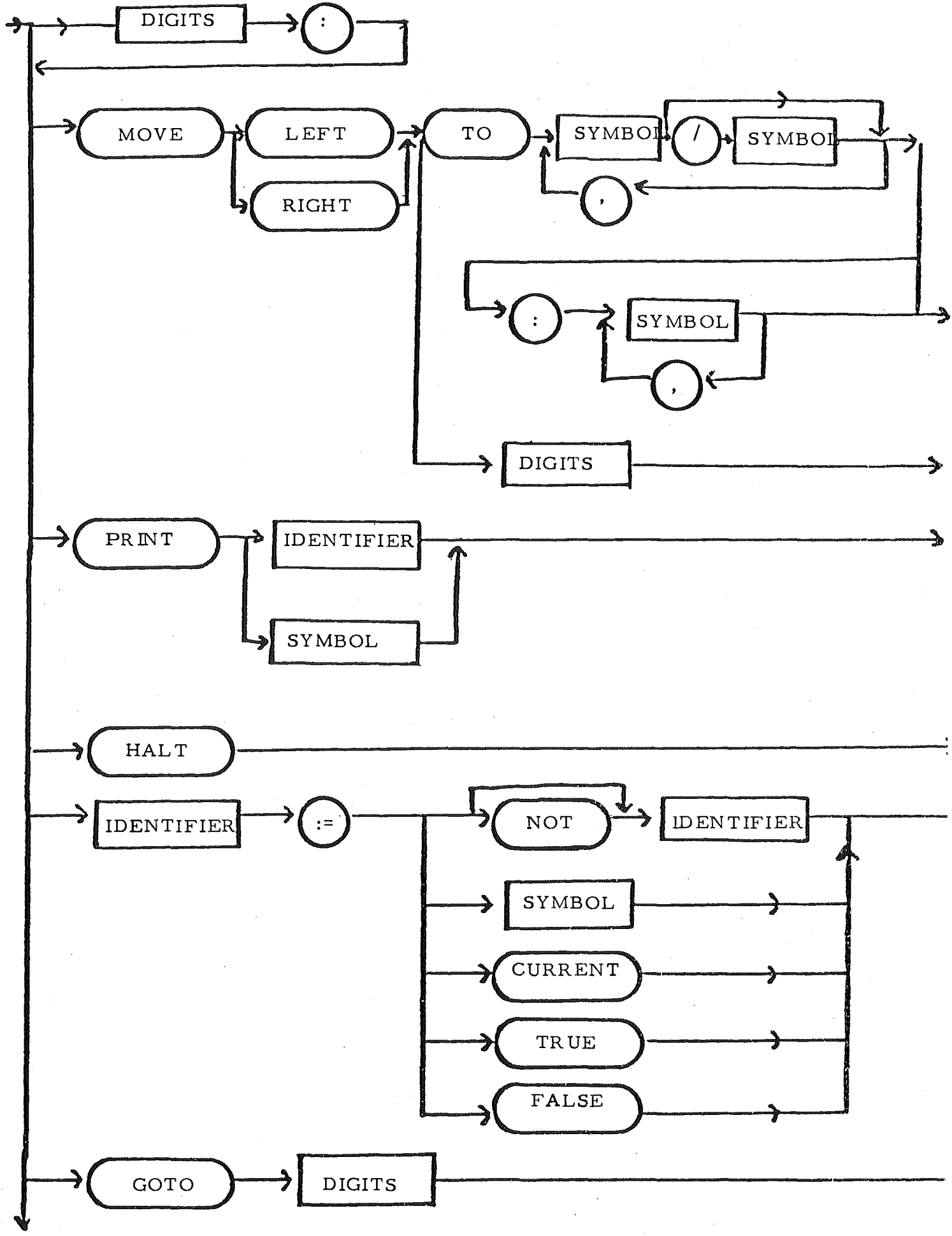
BLOCK



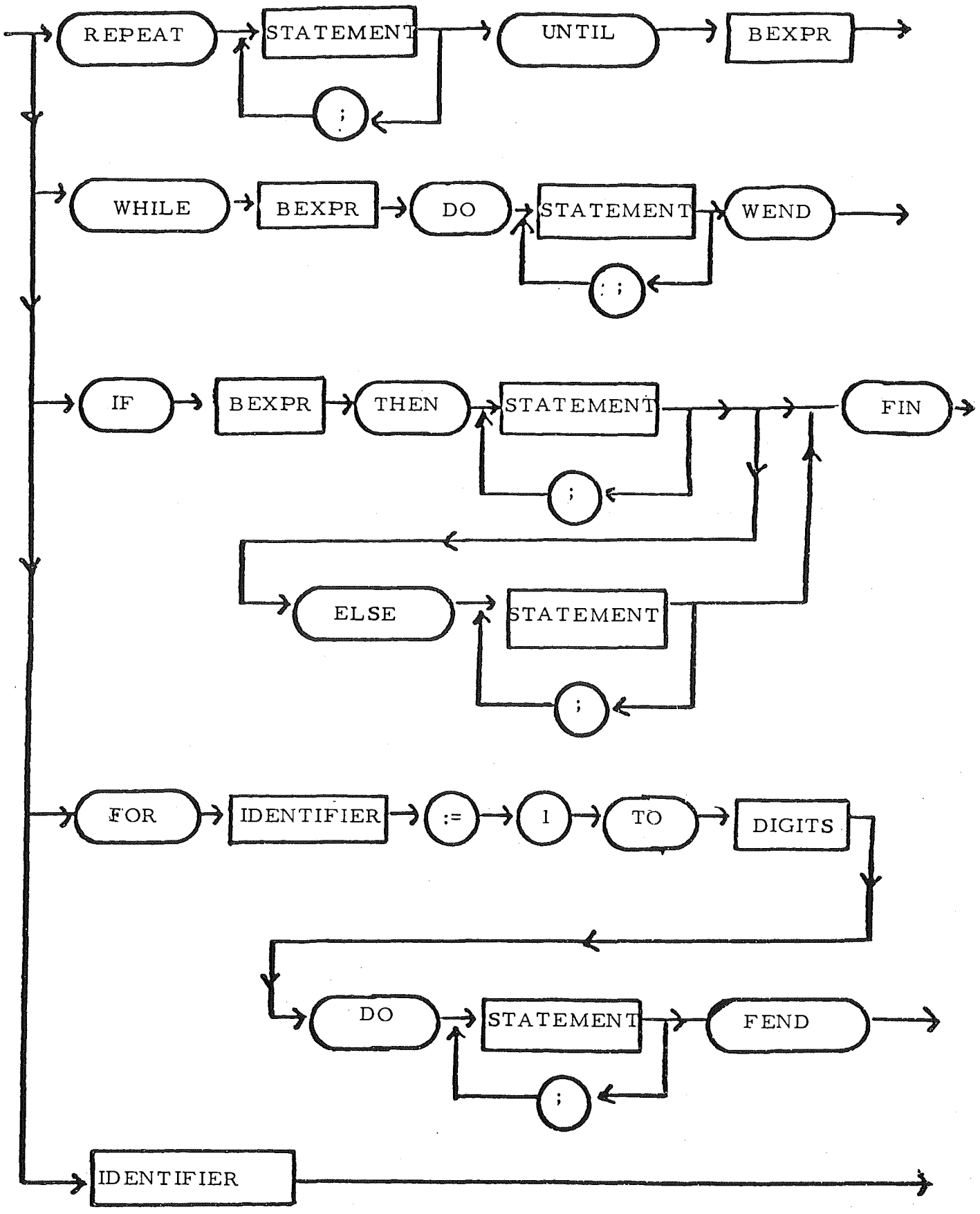
DECLARATIONS



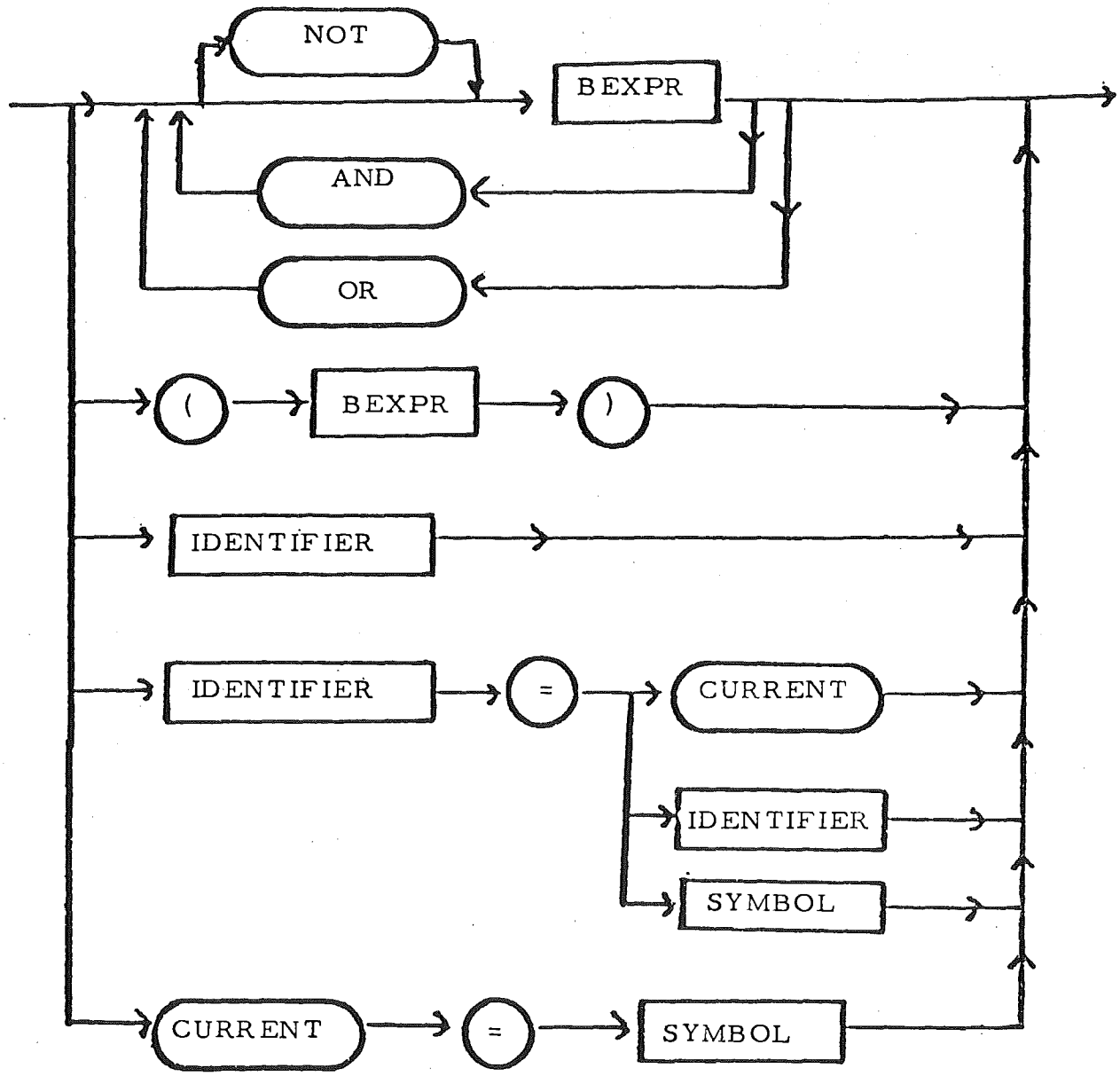
STATEMENT


DIGITS
:
MOVE
LEFT
RIGHT
TO
SYMBOL
/
SYMBOL
,
:
SYMBOL
,
DIGITS
PRINT
IDENTIFIER
SYMBOL
HALT
IDENTIFIER
:=
NOT
IDENTIFIER
SYMBOL
CURRENT
TRUE
FALSE
GOTO
DIGITS

STATEMENT (continued)

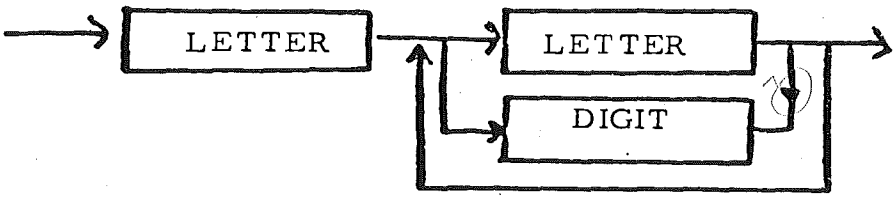


BEXPR

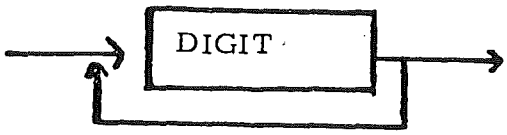


SYMBOL

IDENTIFIER



DIGITS



APPENDIX B

The following is a table of tokens and the associated integer codes used by the compiler.

AND	1	RIGHT	27
ARE	2	SET	28
BOOLEAN	3	START	29
CHAR	4	SYMBOLS	30
CURRENT	4	TAPE	31
DO	6	THEN	32
ELSE	7	TO	33
END	8	TRUE	34
EXTERNAL	9	UNTIL	35
FALSE	10	VAR	36
FEND	11	WEND	37
FIN	12	WHILE	38
FOR	13	: =	39
GOTO	14	:	40
HALT	15	,	41
IF	16	.	42
IN	17	;	43
LABEL	18	=	44
LEFT	19	/	45
MOVE	20	(	46
NOT	21	)	47
OF	22	'	48
OR	23	Identifier	49
PRINT	24	Symbol	50
PROCEDURE	25	Digits	51
REPEAT	26		

APPENDIX C

The following three pages contain examples of two Turingol programs which both perform unary to binary conversion.

```
 1: PROCEDURE UNARYBIN;
 2:
 3:
 4: % THIS PROCEDURE CONVERTS A STRING OF '1' SYMBOLS
 5: % ( A UNARY NUMBER ) TO ITS BINARY EQUIVALENT WHICH IS
 6: % A STRING OF 'A' AND 'B' SYMBOLS.
 7: % E.G.  $1111111$  BECOMES  $BBA$XXXXXX $   (I.E. 6 )
 8:
 9:
10: TAPE SYMBOLS ARE '$' , '1' , ' ' , 'X' , 'A' , 'B';
11: BOOLEAN FLAG:TRUE , AGAIN:FALSE;
12: START
13: REPEAT
14:    AGAIN:=FALSE;
15:    REPEAT
16:       FLAG:=TRUE;
17:       MOVE RIGHT TO '1' / 'X' , '$' ;
18:       IF CURRENT = 'X' THEN
19:          AGAIN:=TRUE;
20:          MOVE RIGHT TO '1','$' : 'A','B',' ';
21:          FLAG:=FALSE FIN
22:    UNTIL CURRENT = '$' ;
23:    IF AGAIN THEN
24:       MOVE LEFT TO ' ';
25:       IF FLAG THEN PRINT 'A'
26:               ELSE PRINT 'B' FIN;
27:       MOVE RIGHT TO '$' : 'X' , '1' , ' '
28:    FIN
29: UNTIL NOT AGAIN
30: END.
```

```
COMPILATION SUCCESSFUL.
NUMBER OF STATES PRODUCED =  8
```

```
 99:
100:
101:
102:        1 $      2 $ 1
103:        1 1      2 1 1
104:        1        2   1
105:        1 X      2 X 1
106:        1 A      2 A 1
107:        1 B      2 B 1
108:        2        2   1
109:        2 X      2 X 1
110:        2 A      2 A 1
111:        2 B      2 B 1
112:        2 $      0 $ 6
113:        2 1      3 X 1
114:        3 X      3 X 1
115:        3 $      4 $ 0
116:        3 1      6 1 1
117:        4 $      4 $ 0
118:        4 1      4 1 0
119:        4 X      4 X 0
120:        4 A      4 A 0
121:        4 B      4 B 0
122:        4        5 B 1
123:        5 A      5 A 1
124:        5 B      5 B 1
125:        5 $      2 $ 1
126:        6        6   1
127:        6 X      6 X 1
128:        6 A      6 A 1
129:        6 B      6 B 1
130:        6 $      7 $ 0
131:        6 1      3 X 1
132:        7 $      7 $ 0
133:        7 1      7 1 0
134:        7 X      7 X 0
135:        7 A      7 A 0
136:        7 B      7 B 0
137:        7        8 A 1
138:        8 A      8 A 1
139:        8 B      8 B 1
140:        8 $      2 $ 1
141:        0 Z      8 Z 0
142:      101
143:      107
```

```
 1: PROCEDURE UNARYBIN2;
 2:
 3:
 4: % THIS PROCEDURE CONVERTS A STRING OF '1' SYMBOLS TO A
 5: % BINARY EQUIVALENT WHICH IS A STRING OF 'A' AND 'B' SYMBOLS.
 6: % E.G.  $111111$  BECOMES  BBA$XXXXXX $  (I.E. 6)
 7:
 8:
 9: TAPE SYMBOLS ARE '$' , '1' , ' ' , 'X' , 'A' , 'B';
10: LABEL 10,20;
11: START
12: 10: MOVE RIGHT TO '1'/'X' , '$' : ' ' , 'A' , 'B';
13:     IF CURRENT = 'X' THEN
14:     20: MOVE RIGHT TO '1' , '$' : ' ' , 'A' , 'B';
15:         IF CURRENT = '1' THEN GOTO 10
16:         ELSE MOVE LEFT TO ' '/'B' FIN
17:     ELSE MOVE LEFT TO ' '/'A' FIN;
18:     MOVE RIGHT TO '1'/'X' , ' ';
19:     IF CURRENT = 'X' THEN GOTO 20 FIN
20: END.
```

```
COMPILATION SUCCESSFUL.
NUMBER OF STATES PRODUCED =  6
```

150		0				
151	1	$	2	$	1	
152	1	1	2	1	1	
153	1		2		1	
154	1	X	2	X	1	
155	1	A	2	A	1	
156	1	B	2	B	1	
157	2	X	2	X	1	
158	2	$	3	$	0	
159	2	1	5	X	1	
160	3	$	3	$	0	
161	3	1	3	1	0	
162	3	X	3	X	0	
163	3	A	3	A	0	
164	3	B	3	B	0	
165	3		4	A	1	
166	4	$	4	$	1	
167	4	X	4	X	1	
168	4	A	4	A	1	
169	4	B	4	B	1	
170	4	1	5	X	1	
171	4		0		6	
172	5	X	5	X	1	
173	5	$	6	$	0	
174	5	1	2	1	1	
175	6	$	6	$	0	
176	6	1	6	1	0	
177	6	X	6	X	0	
178	6	A	6	A	0	
179	6	B	6	B	0	
180	6		4	B	1	
181	0	Z	6	Z	0	
182	150					
183	156					
184	159					
185	165					
186	171					
187	174					
188	0					

TURINGOL TURING MACHINE SIMULATOR

PROGRAM NAME : UNARYBIN1

INPUT TAPE
******
$111111111111111111111111111111111111111111111111111111111$
******
END OF TAPE

OUTPUT TAPE
******
BBAABA$XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX$
******
END OF TAPE

TURING MACHINE HALTS IN STATE 2

****

TURINGOL TURING MACHINE SIMULATOR

PROGRAM NAME : UNARYBIN2

INPUT TAPE
******
$111111111111111111111111111111111111111111111111111111111$
******
END OF TAPE

OUTPUT TAPE
******
BBAABA$XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX$
******
END OF TAPE

TURING MACHINE HALTS IN STATE 4

****

BIBLIOGRAPHY

1 Computation Theory. N. D. Jones 1973.

2 Journal of the Association for Computing Machinery. Vol. 14, No. 4, Oct. 1967, pp 615-635. Programming Languages for Automata. D. Knuth, R. Bigelow.

3 Journal of the Assotiation for Computing Machinery 1965. A Turing Machine Simulator. M. W. Curtis.

4 Mathematical Systems Theory. Vol. 2, No. 2. Semantics of Context Free Languages. D. Knuth.